as

The GNU Assembler

Dean Elsner, Jay Fenlason & friends

Sept 87

The Free Software Foundation Inc. thanks The Nice Computer Company of Australia for loaning Dean Elsner to write the first (Vax) version of as for Project GNU. The proprietors, management and staff of TNCCA thank FSF for distracting the boss while they got some work done.

Copyright © 1986,1987 Free Software Foundation, Inc.

Permission is granted to make and distribute verbatim copies of this manual provided the copyright notice and this permission notice are preserved on all copies.

Permission is granted to copy and distribute modified versions of this manual under the conditions for verbatim copying, provided that the entire resulting derived work is distributed under the terms of a permission notice identical to this one.

Permission is granted to copy and distribute translations of this manual into another language, under the same conditions as for modified versions.

Summary Table of Contents

1. Overview, Usage 1
2. Syntax 13
3. (Sub)Segments & Relocation 19
4. Symbols 24
5. Expressions 28
6. Assembler Directives 31
7. Machine Dependent Features 39
8. Maintaining the Assembler 44

Table of Contents

1. Overview, Usage 1
1.1 Notation 1
1.2 Concept of an Assembler 3
1.2.1 What is a *program*? 3
1.2.2 What is Machine Language? 3
1.2.3 Why prefer assembly language over machine language? 4
1.2.4 What is a linker? 6
1.2.5 Why do people use linkers? 6
1.3 as, the GNU Assembler 6
1.4 Command Line Synopsis 7
1.4.1 Switches 7
1.5 Input File(s) 8
1.5.1 Input Filenames and Line-numbers 9
1.6 Output (Object) File 9
1.7 Error and Warning Messages 9
1.8 Optional Switches 10
1.8.1 -f Works Faster 10
1.8.2 -G Includes GDB Symbolic Information 10
1.8.3 -L Includes Local Labels 11
1.8.4 -o Names the Object File 11
1.8.5 -R Folds Data Segment into Text Segment 11
1.8.6 -W Represses Warnings 11
1.8.7 Useless (but Compatible) Switches 12

2. Syntax 13
2.1 The Pre-processor 13
2.2 Whitespace 13
2.3 Comments 14
2.4 Symbols 14
2.5 Statements 15
2.6 Constants 15
2.6.1 Character Constants 16
2.6.1.1 Strings 16
2.6.1.2 Characters 17
2.6.2 Number Constants 17
2.6.2.1 Integers 17
2.6.2.2 Bignums 17
2.6.2.3 Flonums 18

3. (Sub)Segments & Relocation 19

3.1 Segments 20
3.1.1 ld segments 20
3.1.2 Mythical Segments 21
3.2 Sub-Segments 22
3.3 Bss Segment 23

4. Symbols 24
4.1 Labels 24
4.2 Giving Symbols Other Values 24
4.3 Symbol Names 24
4.3.1 Local Symbol Names 24
4.4 Symbol Attributes 25
4.4.1 Value 26
4.4.2 Type 26
4.4.2.1 N_EXT bit 26
4.4.2.2 N_TYPE bits 27
4.4.2.3 N_STAB bits 27
4.4.3 Desc(riptor) 27
4.4.4 Other 27

5. Expressions 28
5.1 Empty Expressions 28
5.2 Integer Expressions 28
5.2.1 Primaries 28
5.2.2 Operators 29
5.2.3 Unary Operators 29
5.2.4 Binary Operators 29

6. Assembler Directives 31
6.1 .abort 31
6.2 .align *absolute-expression* , *absolute-expression* 31
6.3 .ascii *strings* 31
6.4 .asciz *strings* 31
6.5 .byte *expressions* 32
6.6 .comm *symbol* , *length* 32
6.7 .data *subsegment number* 32
6.8 .desc *symbol*, *absolute-expression* 32
6.9 .double *flonums* 32
6.10 .file *string* 32
6.11 .fill *repeat* , *size* , *value* 33
6.12 .float *flonums* 33
6.13 .gdbbeg *absolute-expression* 33
6.14 .gdbblock *block-number* , *offset* 33

6.15 .gdbend *absolute-expression* . . . 34
6.16 .gdbsym *symbol* , *offset* . . . 34
6.17 .global *symbol* . . . 34
6.18 .int *expressions* . . . 34
6.19 .lcomm *symbol* , *length* . . . 34
6.20 .line *logical line number* . . . 35
6.21 .long *expressions* . . . 35
6.22 .lsym *symbol*, *expression* . . . 35
6.23 .octa *bignums* . . . 35
6.24 .org *new-lc* , *fill* . . . 35
6.25 .quad *bignums* . . . 36
6.26 .set *symbol*, *expression* . . . 36
6.27 .short *expressions* . . . 36
6.28 .space *size* , *fill* . . . 36
6.29 .stabd, .stabn, .stabs . . . 36
6.30 .text *subsegment number* . . . 37
6.31 .word *expressions* . . . 38
6.32 Deprecated Directives . . . 38

7. Machine Dependent Features . . . 39
7.1 Vax . . . 39
7.1.1 Floating Point . . . 39
7.1.2 Machine Directives . . . 39
7.1.2.1 .dfloat *flonums* . . . 39
7.1.2.2 .ffloat *flonums* . . . 40
7.1.2.3 .gfloat *flonums* . . . 40
7.1.2.4 .hfloat *flonums* . . . 40
7.1.3 Opcodes . . . 40
7.1.4 Branch Improvement . . . 40
7.1.5 operands . . . 42
7.2 68020 . . . 42
7.2.1 Floating Point . . . 42
7.2.2 Machine Directives . . . 43
7.2.3 Opcodes . . . 43
7.3 32xxx . . . 43

8. Maintaining the Assembler . . . 44
8.1 Design . . . 44
8.2 Deprecated Feature(?)s . . . 45
8.3 Bugs, Ideas, Further Work . . . 45
8.4 Sources . . . 46
8.4.1 Test Directory . . . 49

8.4.1.1 Regression Testing 50

1. Overview, Usage

This document describes the GNU assembler **as**. Only experienced programmers should read it, because using an assembler requires detailed understanding of a computer's representation of programs as binary numbers.

- The concept of assembly is defined.
- The invocation of the **as** program is described.
- Legal assembler statements are described, with semantic examples.
- Statements provided for particular computer architectures are described, with examples.
- Assembler internals are described, for people who want to maintain the **as** program.

Throughout this document, we assume that you are running *GNU*, the portable operating system from the *Free Software Foundation, Inc.*. This restricts our attention to certain kinds of computer; once this assumption is granted examples and definitions need less qualification.

Readers should already comprehend:

- Von Neumann computer architecture
- Central processing unit, containing registers
- 32-bit microcomputer
- memory address
- contents of memory address
- bit
- 8-bit byte
- 2's complement arithmetic

as is part of a team of programs that produce some numbers that a computer can interpret as a program. Different versions of **as** are used for different kinds of computer. These ideas are detailed later.

1.1 Notation

GNU and **as** assume the computer that will run the programs it assembles will obey these rules.

A (memory) *address* is 32 bits. The lowest address is zero.

The *contents* of any memory address is one *byte* of exactly 8 bits.

A *word* is 16 bits composed of two bytes. A word is stored in two bytes of memory. The addresses of the bytes differ by exactly 1. Notice that the interpretation of the bits in a word and of how to address a word depends on which particular computer you are assembling for.

A *long word*, or *long*, is 32 bits composed of four bytes. It is stored in 4 bytes of memory; these bytes have contiguous addresses. Again the interpretation and addressing of those bits is machine dependent. 32xxx computers say *double word* where we say *long*.

Most documentation for

Numeric quantities are usually *unsigned* or *2's complement*. Bytes, words and longs may store numbers. `as` manipulates integer expressions as 32-bit numbers in 2's complement format. When asked to store an integer in a byte or word, the lowest order bits are stored. The order of bytes in a word or long in memory is determined by what kind of computer will run the assembled program. We won't mention this important *caveat* again.

Sorry about the strange use of some of these terms. Although *byte* used to mean any length of contiguous bits, *byte* now pervasively means exactly 8 contiguous bits. A *word* of 16 bits made sense for 16-bit computers. Even on 32-bit computers, a *word* still means 16 bits (to machine language programmers). To many other programmers of GNU a *word* means 32 bits, so beware. Similarly *long* means 32 bits: from "long word". 32xxx machine language calls a 32-bit number a "double word".

delete.

Names for integers of different sizes: some conventions

length (bits)	as	vax	32xxx	68020	GNU C
8	byte	byte	byte	byte	char
16	word	word	word	word	short (int)
32	long	long(-word)	double-word	long(-word)	long (int)
64	quad	quad(-word)			
128	octa	octa-word			

1.2 Concept of an Assembler

Here is an example of what an assembler does, developing through this section, showing:

- What is machine language? How does a machine language program get performed?
- Why do people prefer to use assembly language instead of machine language?
- What is a linker?
- Why do people use linkers?

1.2.1 What is a *program*?

A program is an ordered sequence of events. Think of a concert program for example. All players in the orchestra are expected to play the same piece at the same time, or the results are catastrophic. Further most music doesn't make sense if it is played too slow, too fast, or backwards.

Computer programs do not often command musical sounds. Usually they manipulate numbers. Often the numbers are in turn a code for letters and punctuation. Like concert programs however, most programs are designed in modular parts, and within each part it is vital that the right thing is done at the right time.

Programs can be represented in many ways. When a computer is obeying a program then it is represented electromagnetically in the semiconductors and metals of the computer's *central processor*. When a program resides in the computer's memory it is stored as numbers, because that is all a computer's memory can remember: just numbers. When programmers talk about and write programs, they usually write the programs out using letters and numbers in much the same way as people write recipies for cooking.

1.2.2 What is Machine Language?

Machine language is numbers. Each different kind of computer has a different machine language: has a different interpretation of numbers as computer instructions. A computer operates by fetching a number from it's memory, interpreting it as an instruction to do something, doing it, then fetching the 'next' number from memory.

There is a convention for each kind of computer that determines where the 'next' instruction may be found in the computer's memory. This convention also determines what the computer will

do given any particular number as the 'next' instruction. The convention describes different states of the computer's registers, memory and peripherals, and how the computer changes from one state to the next state. This convention is called the *machine language* of the computer.

For example, let's pretend that the numbers '`42, 69, 105, 288`' are, by the rules of our computer's design, an instruction to add the 14th number in the computer's memory to the 10th register in the computer's 'central processor'.

1.2.3 Why prefer assembly language over machine language?

Assembly language tries to represent all those machine language numbers as (more mnemonic) symbols. Assembly language still talks about the same registers, memories and peripherals as machine language. Because different computers have different machine languages, different computers have different assembly languages.

The machine language instruction '`42, 69, 105, 288`' might be written in assembly language as '`add money, interest`'. Most people find the assembly language easier to read, easier to speak over a telephone and faster to understand.

There are, broadly, two kinds of symbols used in this sort of assembler: *permanent symbols* and *user symbols*.

Permanent symbols mean the same thing in any program. For example, '`add`' might always mean "add the contents of a memory cell to the contents of a register, replacing that register's contents with the sum".

User symbols can mean different things in different assembly language programs. You must tell the assembler what a user symbol represents in the same program as you use that symbol. Symbols are often defined in terms of other user symbols, but somewhere you must define at least one user symbol using numbers, permanent symbols or concepts built into the assembler.

Notice that for user symbols, the programmer chooses their names. This is important because programmers can use their words that say something about the program's task. Try reading a Swedish program if you think names are unimportant. Carefully chosen symbol names make the meaning of a program clear.

In assembly language you may use *comments* which are texts that humans read and write but have no influence on how the program performs. Most machine languages don't permit comments.

A comment might look like '`/* cents */`'.

In assembly language you can often lay out comments and assembler 'statements' to please the human eye. Machine languages do not permit this. When you read machine language programs all the cognitive power of your sight is unused.

So instead of writing '`42, 69, 105, 288`' many people prefer to write (and read) something like the program fragment

```
money = %6
...
interest:  .block 1  /*in ten thousandths of cents*/
...
add money, interest
...
```

This example might 'tell' the assembler to emit numbers such as '... `42, 69, 144, 96` ...'.

Computers are very good at detecting inconsistencies in machine language programs but they only report them as (you guessed it) numbers! So machine language programmers constantly interconvert numbers and their meanings, which is very tiring and mistake-prone for most people. Assemblers are a little better at reporting inconsistencies, because they can include user symbols in their error messages to the programmer.

An *assembler* is a program that writes other programs. It reads a textual (editable) program and writes out some numbers which capture (a meaning of) the text in a form better suited for a computer to process the program. The numbers written out still need processing by a "linker" program (exampled below) before a computer can execute those numbers as a program.

The assembler does clerical tasks like:

- ignoring comments
- assigning memory addresses (a special kind of number) to user symbols
- ignoring the way a program is placed on the printed page
- converting user and permanent symbols to correct machine language numbers
- a host of bookkeeping tasks too trivial to list here
- giving the programmer informative messages about any inconsistencies that the assembler detects

1.2.4 What is a linker?

A *linker* combines the results of 'assembling' several assembly language (partial) programs, into one whole program.

The linker works out where each partial program will live in the computer's memory when the whole program is running, then prepares the numbers to place in the computer's memory that represent the combined program. The major job of the linker is to resolve the actual addresses the program needs to run.
After the linker has linked the partial programs together into one computer file, the file can be read into the computer's memory and executed.

1.2.5 Why do people use linkers?

By linking your assembled program with other people's assembled programs, you can use their programs. This often saves you the trouble of writing and debugging those programs yourself.

If you use trusted partial programs then when your program is seen not to work you can be far more confident that the mistake is in your partial program than in the other partial programs.

Linkers permit you to refer to other people's programs using symbolic names. The linker enforces conventions that permit many users to write parts of programs, assign names to the parts, and contribute them for use by other programmers.

Because the idea of naming and linking programs has very little to do with the actual numbers in the parts of programs, it is normal for the same linker to work on many different kinds of computer. Assemblers have to know a lot about the machine language they assemble programs into, but linkers have to know very little about the particular machine beyond certain addressing conventions.

1.3 as, the GNU Assembler

As is an assembler; it is one of the team of programs that 'compile' your programs into the binary numbers that a computer uses to 'run' your program. Often `as` reads a *source* program written by a compiler and writes an *object* program for the linker `ld` to read.

The source program consists of *statements* and comments. Each statement might *assemble* to one machine language instruction or to one very simple datum.

Mostly you don't have to think about the assembler because the compiler invokes it as needed; in that sense the assembler is just another part of the compiler. If you write your own assembly language program, then you must run the assembler yourself to get an object file suitable for linking. You can read below how to do this.

The only thing `as` is guaranteed to do is assemble the output of the C compiler `cc` for use by the linker `ld`. `as` (vax version) tries to assemble correctly everything that BSD 4.2 assembler would assemble: except bit-fields and `.org`.

Each version of the assembler knows about just one kind of machine language, but much is common between the versions, including object file formats, (most) assembler directives and assembler syntax.

Unlike older assemblers, `as` tries to assemble a source program in one pass of the source file. This subtly changes the meaning of the `.org` directive (See section 6.24 [Org], page 35.).

If you want to write assembly language programs, you must tell `as` what numbers should be in a computer's memory, and which addresses should contain them, so that the program may be executed by the computer. Using symbols will prevent many bookkeeping mistakes that can occur if you use raw numbers.

1.4 Command Line Synopsis

```
as -G  GDB_symbol_file -L -o object_file -R -W  input1 input2 ...
```

After the program name `as` the command line may contain switches and file names in any order. The order of switches doesn't matter but the order of file names is significant. Only the assembler's name `as` is compulsory and it must (of course) be first.

1.4.1 Switches

Except for '`--`' any command line argument that begins with a dash ('`-`') is a switch. Each switch changes the behavior of `as`. No switch changes the way another switch works. A switch is a

'-' followed by a letter; the case of the letter is important. No switch (letter) should be used twice on the same command line. All switches are optional.

Some switches expect exactly one file name to follow them. The file name may either immediately follow the switch's letter (compatible with older assemblers) or it may be the next command argument (GNU standard). These two command lines are equivalent:

```
as -o my-object-file.o mumble
as -omy-object-file.o mumble
```

Always, '--' (that's two dashes, not one) by itself names the standard input file.

1.5 Input File(s)

We use the words *source program*, abbreviated *source*, to describe the program input to one run of **as**. The program may be in one or more GNU files; how the source is partitioned into files doesn't change the meaning of the source.

The source text is a catenation of the text in each file.

Each time you run **as** it assembles exactly one source program. A source program text is made of one or more GNU files. (The standard input is also a file.)

You give **as** a command line that has zero or more input file names. The input files are read (from left file name to right). A command line argument (in any position) that has no special meaning is taken to be an input file name. If **as** is given no file names it attempts to read one input file from **as**'s standard input.

Use '--' if you need to explicitly name the standard input file in your command line.

It is OK to assemble an empty source. You get a small harmless object (output) file.

If you try to assemble no files then **as** will try to read standard input, which is normally your terminal. You may have to type `CTL-D` to tell **as** there is no more program to assemble.

1.5.1 Input Filenames and Line-numbers

A line is text up to and including the next newline ('\n'). The first line of a file is numbered **1**, the next **2** and so on.

There are two ways of locating a line in the input file(s) and both are used in reporting error messages. One way refers to a line number in a physical file; the other refers to a line number in a logical file.

Physical files are those files named in the command line given to `as`.

Logical files are "pretend" files which bear no relation to physical files. Logical file names help error messages reflect the proper source file. Often they are used when `as`' source is itself synthesized from other files.

1.6 Output (Object) File

Every time you run `as` it produces an output file, which is your assembly language program translated into numbers. This file is the object file; named `a.out` unless you tell `as` to give it another name by using the `-o` switch. Conventionally, object file names end with '`.o`'.

The object file is for input to the linker `ld`. It contains assembled program code, information to help `ld` to integrate the assembled program into a runnable file and notes for the debugger `gdb`. The precise format of object files is described elsewhere.

1.7 Error and Warning Messages

`as` may write warnings and error messages to the standard error file (usually your terminal). This does not happen when `as` is run automatically by a compiler. Error messages are useful for those (few) people who still write in assembly language.

Warnings report an assumption made so that `as` could keep assembling a flawed program.

Errors report a grave problem that stops the assembly.

Warning messages have the format

```
file_name:line_number:Warning Message Text
```

If a logical file name has been given (See section 6.10 [File], page 32.) it is used for the filename, otherwise the name of the current input file is used. If a logical line number was given (See section 6.20 [Line], page 35.) then it is used to calculate the number printed, otherwise the actual line in the current source file is printed. The message text is intended to be self explanatory (In the grand UN*X tradition).

Error messages have the format

```
file_name:line_number:FATAL:Error Message Text
```

The file name and line number are derived the same as for warning messages. The actual message text may be rather less explanatory because many of them aren't supposed to happen.

1.8 Optional Switches

1.8.1 -f Works Faster

'`-f`' should only be used when assembling programs written by a (trusted) compiler. It tells `as` that it doesn't have to pre-process the source files.

1.8.2 -G Includes GDB Symbolic Information

[[want a name like "gdb symbol segment" but without the the overloaded word "segment"]]

The C compiler may produce (apart from an assembler source file of your program) a `gdb` symbol segment, in a file. Certain assembler statements manipulate this segment, and `as` can include the symbol segment in the object file that is the result of your assembly.

Use this switch to say which file contains the symbol segment. The switch needs exactly one filename.

`as` directives that begin with '`.gdb...`' manipulate this `gdb` symbol segment. Unless you use a '`-G`' switch all '`.gdb...`' assembler statements are ignored.

The `gdb` notes file is described elsewhere.

1.8.3 -L Includes Local Labels

For historical reasons, labels beginning with 'L' (upper case only) are called *local labels*. Normally you don't see such labels because they are intended for the use of programs (like compilers) that compose assembler programs, not for your notice. Normally both `as` and `ld` discard such labels, so you don't normally debug with them.

This switch tells `as` to retain those 'L...' symbols in the object file. Usually if you do this you also tell the linker `ld` to preserve symbols whose names begin with 'L'.

1.8.4 -o Names the Object File

There is always one object file output when you run `as`. By default it has the name '`a.out`'. You use this switch (which takes exactly one filename) to give the object file a different name.

Whatever the object file is called, `as` will overwrite any existing file of the same name.

1.8.5 -R Folds Data Segment into Text Segment

`-R` tells `as` to write the object file as if all data-segment data lives in the text segment. This is only done at the very last moment: your binary data is the same, but data segment parts are relocated differently. The data segment part of your object file is zero bytes long because all it bytes are appended to the text segment.

When you use `-R` it would be nice to generate shorter address displacements (possible because we don't have to cross segments) between text and data segment. We don't do this simply for compatibility with older versions of `as`. `-R` may work this way in future.

1.8.6 -W Represses Warnings

`as` should never give a warning or error message when assembling compiler output. But programs written by people often cause `as` to give a warning that a particular assumption was made. All such

warnings are directed to the standard error file. If you use this switch, any warning is repressed. This switch only affects warning messages: it cannot change any detail of how **as** assembles your file. Errors, which stop the assembly, are still reported.

1.8.7 Useless (but Compatible) Switches

As accepts any of these switches, gives a warning message that the switch was ignored and proceeds. These switches are for compatibility with scripts designed for other people's assemblers.

`-D` (Debug)
`-S` (Symbol Table)
`-T` (Token Trace)

Obsolete switches used to debug old assemblers.

`-V` (Virtualize Interpass Temporary File)

Other assemblers use a temporary file. This switch commanded them to keep the information in active memory rather than in a disk file. **as** always does this, so this switch is redundant.

`-J` (JUMPify Longer Branches)

Many 32-bit computers permit a variety of branch instructions to do the same job. Some of these instructions are short (and fast) but have a limited range; others are long (and slow) but can branch anywhere in virtual memory. Often there are 3 flavors of branch: short, medium and long. Other assemblers would emit short and medium branches, unless told by this switch to emit short and long branches. This is an archaic machine-dependent switch.

`-d` (Displacement size for JUMPs)

Like the `-J` switch, this is archaic. It expects a number following the `-d`. Like switches that expect filenames, the number may immediately follow the `-d` (old standard) or constitute the whole of the command line argument that follows `-d` (GNU standard).

`-t` (Temporary File Directory)

Other assemblers may use a temporary file, and this switch takes a filename being the directory to site the temporary file. **as** does not use a temporary disk file, so this switch makes no difference. `-t` needs exactly one filename.

2. Syntax

This chapter informally defines most of the syntax allowed in a source file. `as` has ordinary syntax; it tries to be upward compatible from BSD 4.2 assembler except `as` does not assemble Vax bit-fields.

2.1 The Pre-processor

The preprocess phase handles several aspects of the syntax. Use of the pre-processor can be controlled by the '`-f`' option, and by the special comments `#APP` and `#NO_APP`. The option to disable the pre-processor was designed to make compiler output assemble as fast as possible.

The pre-processor adjusts and removes extra whitespace. It leaves one space or tab before the keywords on a line, and turns any other whitespace on the line into a single space.

The pre-processor removes all comments, replacing them with a single space (for /* ... */ comments), or an appropriate number of newlines.

The pre-processor converts character constants into the appropriate numeric values.

This means that excess whitespace, comments, and character constants cannot be used in the portions of the input text that are not pre-processed.

If the first line of an input file is `#NO_APP` or the '`-f`' option is given, the input file will not be pre-processed. Within such an input file, parts of the file can be pre-processed by putting a line that says `#APP` before the text that should be pre-processed, and putting a line that says `#NO_APP` after them. This feature is mainly intend to support asm statements in compilers whose output normally does not need to be pre-processed.

2.2 Whitespace

Whitespace is one or more blanks or tabs, in any order. Whitespace is used to separate symbols, and to make programs neater for people to read. Unless within character constants (See section 2.6.1 [Characters], page 16.), any whitespace means the same as exactly one space.

2.3 Comments

There are two ways of rendering comments to **as**. In both cases the comment is equivalent to one space.

Anything from '`/*`' to the next '`*/`' inclusive is a comment.

```
/*
  The only way to include a newline ('\n') in a comment
  is to use this sort of comment.
*/
/* This sort of comment does not nest. */
```

Anything from the *line comment* character to the next newline considered a comment and is ignored. The line comment character is '`#`' on the Vax, and '`|`' on the 68020. See chapter 7 [MachineDependent], page 39.

To be compatible with past assemblers a special interpretation is given to lines that begin with '`#`'. Following the '`#`' an absolute expression (see chapter 5 [Expressions], page 28) is expected: this will be the logical line number of the **next** line. Then a string (See section 2.6.1.1 [Strings], page 16.) is allowed: if present it is a new logical file name. The rest of the line, if any, should be whitespace.

If the first non-whitespace characters on the line are not numeric, the line is ignored. (Just like a comment.)

```
                          # This is an ordinary comment.
# 42-6 "new_file_name"    # New logical file name
                          # This is logical line # 36.
```

This feature is deprecated, and may disappear from future versions of **as**.

2.4 Symbols

A *symbol* is one or more characters chosen from the set of all letters (both upper and lower case), digits and the three characters '`_.$`'. No symbol may begin with a digit. Case is significant. There is no length limit: all characters are significant. Symbols are delimited by characters not in that set, or by begin/end-of-file. (See chapter 4 [Symbols], page 24.)

2.5 Statements

A *statement* ends at a newline character ('\n') or at a semicolon (';'). The newline or semicolon is considered part of the preceding statement. Newlines and semicolons within character constants are an exception: they don't end statements. It is an error to end any statement with end-of-file: the last character of any input file should be a newline.

You may write a statement on more than one line if you put a backslash (\) immediately in front of any newlines within the statement. When `as` reads a backslashed newline both characters are ignored. You can even put backslashed newlines in the middle of symbol names without changing the meaning of your source program.

An empty statement is OK, and may include whitespace. It is ignored.

Statements begin with zero or more labels, followed by a *key symbol* which determines what kind of statement it is. The key symbol determines the syntax of the rest of the statement. If the symbol begins with a dot (.) then the statement is an assembler directive: typically valid for any computer. If the symbol begins with a letter the statement is an assembly language *instruction*: it will assemble into a machine language instruction. Different versions of `as` for different computers will recognize different instructions. In fact, the same symbol may represent a different instruction in a different computer's assembly language.

A label is usually a symbol immediately followed by a colon (:). Whitespace before a label or after a colon is OK. You may not have whitespace between a label's symbol and its colon. Labels are explained below. See section 4.1 [Labels], page 24.

```
label:      .directive    followed by something
another$label:            # This is an empty statement.
            instruction   operand_1, operand_2, ...
```

2.6 Constants

A constant is a number, written so that its value is known by inspection, without knowing any context. Like this:

```
.byte  74, 0112, 092, 0x4A, 0X4a, 'J, '\J # All the same value.
.ascii "Ring the bell\7"                  # A string constant.
.octa  0x123456789abcdef0123456789ABCDEF0 # A bignum.
```

```
.float 0f-314159265358979323846264338327\
95028841971.693993751E-40                  # - pi, a flonum.
```

2.6.1 Character Constants

There are two kinds of character constants. *Characters* stand for one character in one byte and their values may be used in numeric expressions. String constants (properly called string *literals*) are potentially many bytes and their values may not be used in arithmetic expressions.

2.6.1.1 Strings

A *string* is written between double-quotes. It may contain double-quotes or null characters. The way to get weird characters into a string is to *escape* these characters: precede them with a backslash (\) character. For example '\\' represents one backslash: the first \ is an escape which tells **as** to interpret the second character literally as a backslash (which prevents **as** from recognizing the second \ as an escape character). The complete list of escapes follows.

`\EOF` To have a \ followed by end-of-file is an error. It is treated just like an end-of-file without a preceding backslash.

`\b` Mnemonic for backspace; for ASCII this is octal code 010.

`\f` Mnemonic for FormFeed; for ASCII this is octal code 014.

`\n` Mnemonic for newline; for ASCII this is octal code 012.

`\r` Mnemonic for carriage-Return; for ASCII this is octal code 015.

`\t` Mnemonic for horizontal Tab; for ASCII this is octal code 011.

\ *digit digit digit*
An octal character code. The numeric code is 3 octal digits. For compatibility with other Un*x systems, 8 and 9 are legal digits with values 010 and 011 respectively.

`\\` Represents one '\' character.

`\"` Represents one '"' character. Needed in strings to represent this character, because an unescaped '"' would end the string.

\ *anything-else*
Any other character when escaped by \ will give a warning, but assemble as if the '\' was not present. The idea is that if you used an escape sequence you clearly didn't want the literal interpretation of the following character. However **as** has no other interpretation, so **as** knows it is giving you the wrong code and warns you of the fact.

Which characters are escapable, and what those escapes represent, varies widely among assemblers. The current set is what we think BSD 4.2 `as` recognizes, and is a subset of what most C compilers recognize. If you are in doubt, don't use an escape sequence.

2.6.1.2 Characters

A single character may be written as a single quote immediately followed by that character. The same escapes apply to characters as to strings. So if you want to write the character backslash, you must write `'\\` where the first `\` escapes the second `\`. As you can see, the quote is an accent acute, not an accent grave. A newline (or semicolon (`;`)) immediately following an accent acute is taken as a literal character and does not count as the end of a statement. The value of a character constant in a numeric expression is the machine's byte-wide code for that character. GNU assumes your character code is ASCII: `'A` means 65, `'B` means 66, and so on.

2.6.2 Number Constants

`as` distinguishes 3 flavors of numbers according to how they are stored in the target machine. *Integers* are numbers that would fit into an `int` in the C language. *Bignums* are integers, but they are stored in a more than 32 bits. *Flonums* are floating point numbers, described below.

2.6.2.1 Integers

An octal integer is '`0`' followed by zero or more of the octal digits '`01234567`'.

A decimal integer starts with a non-zero digit followed by zero or more digits ('`0123456789`').

A hexadecimal integer is '`0x`' or '`0X`' followed by one or more hexadecimal digits chosen from '`0123456789abcdefABCDEF`'.

Integers have the obvious values. To denote a negative integer, use the unary operator '`-`' discussed under expressions (See section 5.2.4 [Unops], page 29.).

2.6.2.2 Bignums

A *bignum* has the same syntax and semantics as an integer except that the number (or its

negative) takes more than 32 bits to represent in binary. The distinction is made because in some places integers are permitted while bignums are not.

2.6.2.3 Flonums

A *flonum* represents a floating point number. The translation is complex: a decimal floating point number from the text is converted by `as` to a generic binary floating point number of more than sufficient precision. This generic floating point number is converted to the particular computer's floating point format(s) by a portion of `as` specialized to that computer.

A flonum is written by writing (in order)

- The digit '`0`'.
- A letter, to tell `as` the rest of the number is a flonum. `e` is recommended. Case is not important. (Any otherwise illegal letter will work here, but that might be changed. BSD 4.2 assembler seems to allow any of '`defghDEFGH`'.)
- An optional sign: either '`+`' or '`-`'.
- An optional integer part: zero or more decimal digits.
- An optional fraction part: '`.`' followed by zero or more decimal digits.
- An optional exponent, consisting of:
  - A letter; the exact significance varies according to the computer that executes the program. `as` accepts any letter for now. Case is not important.
  - Optional sign: either '`+`' or '`-`'.
  - One or more decimal digits.

At least one of *integer part* or *fraction part* must be present. The floating point number has the obvious value.

The computer running `as` needs no floating point hardware. `as` does all processing using integers.

3. (Sub)Segments & Relocation

Roughly, a *segment* is a range of addresses, with no gaps, with all data "in" those addresses being treated the same. For example a "read only" segment.

The linker `ld` reads many object files (partial programs) and combines their contents to form a runnable program. When `as` emits an object file, the partial program is assumed to start at address 0. `ld` will assign the final addresses the partial program occupies, so that different partial programs don't overlap. That explanation is too simple, but it will suffice to explain how `as` works.

`ld` moves blocks of bytes of your program to their run-time addresses. These blocks slide to their run-time addresses as rigid units; their length does not change and neither does the order of bytes within them. Such a rigid unit is called a *segment*. Assigning run-time addresses to segments is called *relocation*. It includes the task of adjusting mentions of object-file addresses so they refer to the proper run-time addresses.

An object file written by `as` has three segments any of which may be empty. These are named *text*, *data* and *bss* segments. Within the object file, the text segment starts at address 0, the data segment follows, last comes the bss segment.

To let `ld` know which data will change when the segments are relocated, and how to change that data, `as` also writes to the object file details of the relocation needed. To perform relocation `ld` must know for each mention of an address in the object file:

- At what address in the object file does this mention of an address begin?
- How long (in bytes) is this mention?
- Which segment does the address refer to? What is the numeric value of (*address - start-address of segment*)?
- Is the mention of an address "Program counter relative"?

In fact, every address `as` ever thinks about is expressed as (*segment + offset into segment*). Further, every expression `as` computes is of this segmented nature: "segment" to `as` is like `type` in *e.g.* Pascal or C or FORTRAN. So *absolute expression* means an expression with segment "absolute" (See section 3.1.1 [LdSegs], page 20.). A *pass1 expression* means an expression with segment "pass1" (See section 3.1.2 [MythSegs], page 21.). I will write a "(segment, offset)" value as `{` segment-name (offset into segment) `}`.

Apart from text, data and bss segments you need to know about the *absolute* segment. When

`ld` mixes partial programs, addresses in the absolute segment remain unchanged. That is, address {absolute 0} is "relocated" to run-time address 0 by `ld`. Although two partial programs' data segments will not overlap addresses after linking, **by definition** their absolute segments will overlap. Address {absolute 239} in one partial program will always be the same address when the program is running as address {absolute 239} in any other partial program.

The idea of segments is extended to the *undefined* segment. Any address whose segment is unknown at assembly time is by definition rendered {undefined (something, unknown yet)}. Since numbers are always defined, the only way to generate an undefined address is to mention an undefined symbol. A reference to a named common block would be such a symbol: its value is unknown at assembly time so it has segment *undefined.*

By analogy the word *segment* is to describe groups of segments in the linked program. `ld` puts all partial program's text segments in contiguous addresses in the linked program. It is customary to refer to the *text segment* of a program, meaning all the addresses of all partial program's text segments. Likewise for data and bss segments.

3.1 Segments

Some segments are manipulated by `ld`; others are invented for use of `as` and have no meaning except during assembly.

3.1.1 ld segments

`ld` deals with just 5 kinds of segments, summarized below.

text segment
data segment

These segments hold your program bytes. `as` and `ld` treat them as separate but equal segments. Anything you can say of one segment is true of the other. When the program is running however it is customary for the text segment to be unalterable: it will contain instructions, constants and the like. The data segment of a running program is usually alterable: C variables would be stored in the data segment for example.

bss segment

This segment is zeroed bytes when your program begins running. It is used for local commons (eg uninitialized static variables in C programs). The length of each partial program's bss segment is important but because it starts as zeroed bytes there is no

need to store explicit zero bytes in the object file. Bss segment was invented to eliminate those explicit zeros from object files.

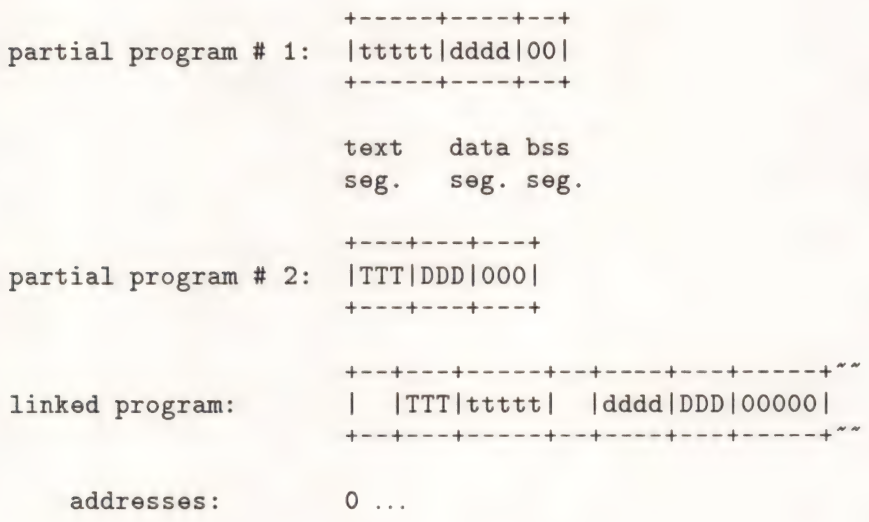
absolute segment

Address 0 of this segment is always "relocated" to runtime address 0. Useful if you want to refer to an address that `ld` must not change when relocating. In this sense we speak of absolute addresses being "unrelocatable": they don't change during relocation.

undefined segment

This "segment" is a catch-all for address references to objects not in the preceding segments. See the description of '`a.out`' for details.

An idealized example of the 3 relocatable segments follows. Memory addresses are on the horizontal axis.



3.1.2 Mythical Segments

These segments are invented for the internal use of `as`. They have no meaning at run-time. You don't need to know about these segments except that they might be mentioned in `as`' warning messages. These segments are invented to permit the value of every expression in your assembly language program to be a segmented address.

absent segment

An expression was expected and none was found.

goof segment

An internal assembler logic error has been found. This means there is a bug in the

assembler.

grand segment

A *grand number* is a bignum or a flonum, but not an integer. If a number can't be written as a C `int` constant, it is a grand number. `as` has to remember that a flonum or a bignum does not fit into 32 bits, and cannot be a primary (See section 5.2.1 [Primary], page 28.) in an expression: this is done by making a flonum or bignum be of type "grand". This is purely for internal `as` convenience; grand segment behaves similarly to absolute segment.

pass1 segment

The expression was impossible to evaluate in the first pass. The assembler will attempt a second pass (second reading of the source) to evaluate the expression. Your expression mentioned an undefined symbol in a way that defies the one-pass (segment + offset in segment) assembly process. No compiler need emit such an expression.

difference segment

As an assist to the C compiler, expressions of the forms

- (undefined symbol) - (expression)
- (something) - (undefined symbol)
- (undefined symbol) - (undefined symbol)

are permitted to belong to the "difference" segment. `as` re-evaluates such expressions after the source file has been read and the symbol table built. If by that time there are no undefined symbols in the expression then the expression assumes a new segment. The intention is to permit statements like '`.word label - base_of_table`' to be assembled in one pass where both `label` and `base_of_table` are undefined. This is useful for compiling C and Algol switch statements, Pascal case statements, FORTRAN computed goto statements and the like.

3.2 Sub-Segments

Assembled bytes fall into two segments: text and data. With each segment, a number of *subsegments*, known as `text 0`, `text 1`, ... `text 8192`, `data 0`, `data 1`, ... `data 8192` are allowed. (Any and all subsegments are optional.)

Each subsegment is zero-padded up to a multiple of four bytes. Subsegments appear in your object file in the above order. (All this to be compatible with other people's assemblers.) The object file, `ld` *etc.* have no concept of subsegments. They just see all your text subsegments as a text segment, and all your data subsegments as a data segment.

To specify which subsegment you want subsequent statements assembled into, use a '`.text` *expression*' or a '`.data` *expression*' statement. *Expression* should be an absolute expression. (See chapter 5 [Expressions], page 28.) If you just say '`.text`' then '`.text 0`' is assumed. Likewise '`.data`' means '`.data 0`'. Assembly begins in `text 0`. For instance:

```
.text 0     # The default subsegment is text 0 anyway.
.ascii "This lives in the first text subsegment. *"
.text 1
.ascii "But this lives in the second text subsegment."
.data 0
.ascii "This lives in the data segment,"
.ascii "in the first data subsegment."
.text 0
.ascii "This lives in the first text segment,"
.ascii "immediately following the asterisk (*)."
```

Each segment has a *location counter* incremented by one for every byte assembled into that segment. Because subsegments are merely a convenience restricted to `as` there is no concept of a subsegment location counter. There is no way to directly manipulate a location counter. The location counter of the segment that statements are being assembled into is said to be the *active* location counter.

3.3 Bss Segment

The `bss` segment is used for local common variable storage. You may allocate address space in the `bss` segment, but you may not dictate data to load into it before your program executes. When your program starts running, all the contents of the `bss` segment are zeroed bytes. Addresses in the bss segment are allocated with a special statement; you may not assemble anything directly into the bss segment. Hence there are no bss subsegments.

4. Symbols

Because the linker uses symbols to link, the debugger uses symbols to debug and the programmer uses symbols to name things, symbols are a central concept. Symbols appear in the object file in the order they are declared. Some debuggers rely on this.

4.1 Labels

A *label* is written as a symbol immediately followed by a colon (':'). The symbol then represents the current value of the active location counter, and is, for example, a suitable instruction operand. You are warned if you use the same symbol to represent two different locations: the first definition overrides any other definitions.

4.2 Giving Symbols Other Values

A symbol can be given an arbitrary value by writing a symbol followed by an equals sign ('=') followed by an expression (see chapter 5 [Expressions], page 28). This is equivalent to using the `.set` directive. (See section 6.26 [Set], page 36.)

4.3 Symbol Names

Symbol names begin with a letter or with one of '`$._`'. That character may be followed by any string of digits, letters, underscores and dollar signs. Case of letters is significant: `foo` is a different symbol name than `Foo`.

Each symbol has exactly one name. Each name in an assembly program refers to exactly one symbol. You may use that symbol name any number of times in an assembly program.

4.3.1 Local Symbol Names

Local symbols help compilers and programmers use names temporarily. There are 10 *local* symbol names, which are re-used throughout the program. Their names are '`0`' '`1`' ... '`9`'. To define a local symbol, write a label of the form *digit*:. To refer to the most recent previous definition of

that symbol write *digit*`b`, using the same digit as when you defined the label. To refer to the next definition of a local label, write *digit*`f` where *digit* gives you a choice of 10 forward references. The '`b`' stands for "backwards" and the '`f`' stands for "forwards".

There is no restriction on how you can use these labels, but remember that at any point in the assembly you can refer to at most 10 prior local labels and to at most 10 forward local labels.

Local symbol names are only a notation device. They are immediately transformed into more conventional symbol names before the assembler thinks about them. The symbol names stored in the symbol table, appearing in error messages and optionally emitted to the object file have these parts:

`L` All local labels begin with '`L`'. Normally both `as` and `ld` forget symbols that start with '`L`'. These labels are used for symbols you are never intended to see. If you give the '`-L`' switch then `as` will retain these symbols in the object file. By instructing `ld` to also retain these symbols, you may use them in debugging.

a digit If the label is written '`0:`' then the digit is '`0`'. If the label is written '`1:`' then the digit is '`1`'. And so on up through '`9:`'.

control-`A` This unusual character is included so you don't accidentally invent a symbol of the same name. The character has ASCII value '`\001`'.

an ordinal number
This is like a serial number to keep the labels distinct. The first '`0:`' gets the number '`1`'; The 15th '`0:`' gets the number '`15`'; *etc.*. Likewise for the other labels '`1:`' through '`9:`'.

For instance, the first `1:` is named `L1^A1`, the 44th `3:` is named `L3^A44`.

4.4 Symbol Attributes

Every symbol has the attributes discussed below. The detailed definitions are in <a.out.h>.

If you use a symbol without defining it, `as` assumes zero for all these attributes, and probably won't warn you. This makes the symbol an externally defined symbol, which is generally what you would want.

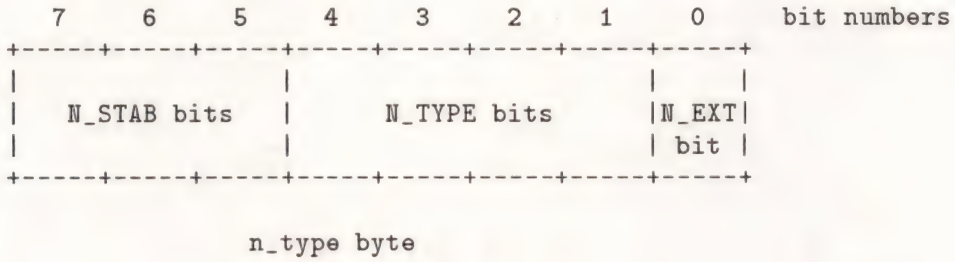
4.4.1 Value

The value of a symbol is (usually) 32 bits, the size of one C `int`. For a symbol which labels a location in the `text`, `data`, `bss` or `Absolute` segments the value is the number of addresses from the start of that segment to the label. Naturally for `text` `data` and `bss` segments the value of a symbol changes as `ld` changes segment base addresses during linking. `absolute` symbols' values do not change during linking: that is why they are called absolute.

The value of an undefined symbol is treated in a special way. If it is 0 then the symbol is not defined in this assembler source program, and `ld` will try to determine its value from other programs it is linked with. You make this kind of symbol simply by mentioning a symbol name without defining it. A non-zero value represents a `.comm` common declaration. The value is how much common storage to reserve, in bytes (*i.e.* addresses). The symbol refers to the first address of the allocated storage.

4.4.2 Type

The type attribute of a symbol is 8 bits encoded in a devious way. We kept this coding standard for compatibility with older operating systems.



n_type byte

4.4.2.1 N_EXT bit

This bit is set if `ld` might need to use the symbol's value and type bits. If this bit is re-set then `ld` can ignore the symbol while linking. It is set in two cases. If the symbol is undefined, then `ld` is expected to find the symbol's value elsewhere in another program module. Otherwise the symbol has the value given, but this symbol name and value are revealed to any other programs linked in the same executable program. This second use of the `N_EXT` bit is most often done by a `.globl` statement.

4.4.2.2 N_TYPE bits

These establish the symbol's "type", which is mainly a relocation concept. Common values are detailed in the manual describing the executable file format.

4.4.2.3 N_STAB bits

Common values for these bits are described in the manual on the executable file format..

4.4.3 Desc(riptor)

This is an arbitrary 16-bit value. You may establish a symbol's descriptor value by using a `.desc` statement (See section 6.8 [Desc], page 32.). A descriptor value means nothing to `as`.

4.4.4 Other

This is an arbitrary 8-bit value. It means nothing to `as`.

5. Expressions

An *expression* specifies an address or numeric value. Whitespace may precede and/or follow an expression.

5.1 Empty Expressions

An empty expression has no operands: it is just whitespace or null. Wherever an absolute expression is required, you may omit the expression and **as** will assume a value of (absolute) 0. This is compatible with other assemblers.

5.2 Integer Expressions

An *integer expression* is one or more *primaries* delimited by *operators*.

5.2.1 Primaries

Primaries are symbols, numbers or subexpressions. Other languages might call primaries "arithmetic operands" but we don't want them confused with "instruction operands" of the machine language so we give them a different name.

Symbols are evaluated to yield {*segment value*} where *segment* is one of **text**, **data**, **bss**, **absolute**, or **undefined**. *value* is a signed 2's complement 32 bit integer. See (???).

Numbers are usually integers (see ???).

A number can be a flonum or bignum (see ????). In this case, you are warned that only the low order 32 bits are used, and **as** pretends these 32 bits are an integer. You may write integer-manipulating instructions that act on exotic constants, compatible with other assemblers.

Subexpressions are a left parenthesis (() followed by an integer expression followed by a right parenthesis ()), or a unary operator followed by an primary.

5.2.2 Operators

Operators are arithmetic marks, like + or %. Unary operators are followed by an primary. Binary operators appear between primaries. Operators may be preceded and/or followed by whitespace.

5.2.3 Unary Operators

`as` has the following *unary operators.* They each take one primary, which must be absolute.

- Dash. *Negation.* Two's complement negation.
~ Tilde. *Complementation.* Bitwise not.

5.2.4 Binary Operators

Binary operators are infix. Operators are prioritized, but equal priority operators are performed left to right. Apart from '+' or '-', both primaries must be absolute, and the result is absolute, else one primary can be either undefined or pass1 and the result is pass1.

1. Highest Priority
   - `*` *Multiplication.*
   - `/` *Division.* Truncation is the same as the C operator '/' of the compiler that compiled `as`.
   - `%` *Remainder.*
   - `<`
   - `<<` *Shift Left.* Same as the C operator '<<' of the compiler that compiled `as`.
   - `>`
   - `>>` *Shift Right.* Same as the C operator '>>' of the compiler that compiled `as`.
2. Intermediate priority
   - `|` *Bitwise Inclusive Or.*
   - `&` *Bitwise And.*
   - `^` *Bitwise Exclusive Or.*
   - `!` *Bitwise Or Not.*
3. Lowest Priority

\+ *Addition.* If either primary is absolute, the result has the segment of the other primary. If either primary is pass1 or undefined, result is pass1. Otherwise + is illegal.

\- *Subtraction.* If the right primary is absolute, the result has the segment of the left primary. If either primary is pass1 the result is pass1. If either primary is undefined the result is difference segment. If both primaries are in the same segment, the result is absolute; provided that segment is one of text, data or bss. Otherwise - is illegal.

The sense of the rules is that you can't add or subtract quantities from two different segments. If both primaries are in one of these segments, they must be in the same segment: **text**, **data** or **bss**, and the operator must be '-'.

6. Assembler Directives

All assembler directives begin with a symbol that begins with a period ('.'). The rest of the symbol is letters: their case does not matter.

6.1 .abort

This directive stops the assembly immediately. It is for compatibility with other assemblers. The original idea was that the assembler program would be piped into the assembler. If the source of program wanted to quit, then this directive tells `as` to quit also. One day `.abort` will not be supported.

6.2 .align *absolute-expression* , *absolute-expression*

Pad the location counter (in the current subsegment) to a word, longword or whatever boundary. The first expression is the number of low-order zero bits the location counter will have after advancement. For example '`.align 3`' will advance the location counter until it a multiple of 8. If the location counter is already a multiple of 8, no change is needed.

The second expression (and the comma) may be omitted: it defaults to '`, 0`'.

If the ',' (and the following *absolute-expression*) are omitted then the second expression is assumed to be {absolute 0} and the padding bytes are zeroed bytes.

6.3 .ascii *strings*

Expects zero or more string literals (See section 2.6.1.1 [Strings], page 16.) separated by commas. Assembles each string (with no automatic trailing zero byte) into consecutive addresses.

6.4 .asciz *strings*

Just like .ascii, but each string is followed by a zero byte. The 'z' in '.asciz' stands for 'zero'.

6.5 .byte *expressions*

Expects zero or more expressions, separated by commas. Each expression is assembled into the next byte.

6.6 .comm *symbol* , *length*

Declares a named common area. Normally `ld` reserves memory addresses for it during linking, so no partial program defines the location of the symbol. Tell `ld` that it must be at least *length* bytes long. `ld` will allocate space that is at least as long as the longest `.comm` request in any of the partial programs linked. *length* is an absolute expression.

6.7 .data *subsegment number*

Tells `as` to assemble the following statements onto the end of the data subsegment numbered *subsegment number* (which is an absolute expression).

6.8 .desc *symbol, absolute-expression*

Set `n_desc` of the symbol to the low 16 bits of *absolute-expression.*

6.9 .double *flonums*

Expect zero or more flonums, separated by commas. Assemble floating point numbers. The exact kind of floating point numbers emitted depends on what computer `as` is assembling for. See the machine-specific part of the manual for the machine the assembler is running on for more information.

6.10 .file *string*

Tells `as` that we are about to start a new logical file. *String* is the new file name. An empty file name is OK, but you must still give the quotes: "". This statement will go away in future: it is only recognized to be compatible with old `as` programs.

6.11 .fill *repeat* , *size* , *value*

result, *size* and *value* are absolute expressions. Emit *repeat* copies of *size* bytes. *Repeat* may be zero or more. *Size* may be zero or more, but if it is more than 8, then it is deemed to have the value 8, compatible with other people's assemblers. The contents of each *repeat* bytes is taken from an 8-byte number. The highest order 4 bytes are zero. The lowest order 4 bytes are *value* rendered in the byte-order of an integer on the computer **as** is assembling for. Each *size* bytes in a repetition is taken from the lowest order *size* bytes of this number. Again, this bizarre behavior is compatible with other people's assemblers.

Size and *value* are optional. If the second comma and *value* are absent, *value* is assumed zero. If the first comma and following tokens are absent, *size* is assumed to be 1.

6.12 .float *flonums*

Expect zero or more flonums, separated by commas. Assemble floating point numbers. The exact kind of floating point numbers emitted depends on what computer **as** is assembling for. See the machine-specific part of the manual for the machine the assembler is running on for more information.

6.13 .gdbbeg *absolute-expression*

Absolute-expression must be at least zero. **as** will remember that a block numbered *absolute-expression* began where the location count is when this statement is read.

6.14 .gdbblock *block-number* , *offset*

Block-number is a **gdb** block number, at least zero, an absolute expression. *Offset* is an offset into the **gdb** symbolic segment [[!!!!]] taken from the file named in the '**-G**' switch; an absolute expression; the lowest offset written by this directive. Two C **int**s are written in the symbolic segment: first the object-file address of the **.gdbbeg** statement of *block number*; then the object-file address of the **.gdbend** statement of *block number*.

6.15 .gdbend *absolute-expression*

Absolute-expression must be at least zero. `as` will remember that a block numbered *absolute-expression* ended where the location count is when this statement is read.

6.16 .gdbsym *symbol* , *offset*

If the '`-G`' switch named a file of `gdb` symbolic segment information then the `n_value` of *symbol* is written as a C `int` starting at *offset* in the symbolic segment [[!!!!]]. *Offset* is an absolute expression. *Symbol* may be defined after the `.gdbsym` statement.

6.17 .global *symbol*

Makes the symbol visible to `ld`. If you define *symbol* in your partial program, its value is made available to other partial programs that are linked with it. Otherwise, *symbol* will take its attributes from a symbol of the same name from another partial program it is linked with.

This is done by setting the `N_EXT` bit of that symbol's `n_type` to 1.

6.18 .int *expressions*

Expect zero or more *expressions*, of any segment, separated by commas. For each expression, emit a 32-bit number that will, at run time, be the value of that expression. The byte order of the expression depends on what kind of computer will run the program.

6.19 .lcomm *symbol* , *length*

Reserve *length* (an absolute expression) bytes for a local common and denoted by *symbol*, whose segment and value are those of the new local common. The addresses are allocated in the `bss` segment, so at run-time the bytes will start off zeroed. *Symbol* is not declared global (See section 6.17 [Global], page 34.), so is normally not visible to `ld`.

6.20 .line *logical line number*

This tells **as** to change the logical line number. *logical line number* is an absolute expression. The next line will have that logical line number. So any other statements on the current line (after a ;) will be reported as on logical line number *logical line number* - 1. One day this directive will be unsupported: it is used only for compatibility with existing assembler programs.

6.21 .long *expressions*

The same as '`.int`', see section 6.18 [Int], page 34.

6.22 .lsym *symbol*, *expression*

Create a new symbol named *symbol*, but do not put it in the hash table, ensuring it cannot be referenced by name during the rest of the assembly. Set the attributes of the symbol to be the same as the expression value. `n_other` = `n_desc` = 0. `n_type` = (whatever segment the expression has); the `N_EXT` bit of `n_type` is zero. `n_value` = (expression's value).

6.23 .octa *bignums*

Expect zero or more bignums, separated by commas. For each bignum, emit an 16-byte (**octa**-word) integer.

6.24 .org *new-lc* , *fill*

This will advance the location counter of the current segment to *new-lc*. *new-lc* is either an absolute expression or an expression with the same segment as the current subsegment. That is, you can't use `.org` to cross segments. Because **as** tries to assemble programs in one pass *new-lc* must be defined. If you really detest this restriction we eagerly await a chance to share your improved assembler. Compatible with former assemblers, if the segment of *new-lc* is absolute then we pretend the segment of *new-lc* is the same as the current subsegment.

Beware that the origin is relative to the start of the segment, not to the start of the subsegment. This is compatible with other people's assemblers.

If the location counter (of the current subsegment) is advanced, the intervening bytes are filled with *fill* which should be an absolute expression. If the comma and *fill* are omitted, *fill* defaults to zero.

6.25 .quad *bignums*

Expect zero or more bignums, separated by commas. For each bignum, emit an 8-byte (**quad**-word) integer. If the bignum won't fit in a quad-word, warn; just take the lowest order 8 bytes of the bignum.

6.26 .set *symbol*, *expression*

Set the value of *symbol* to expression. This will change `n_value` and `n_type` to conform to the *expression*.

It is OK to `.set` a symbol many times in the same assembly. If the expression's segment is unknowable during pass 1, a second pass over the source program will be forced.

6.27 .short *expressions*

The same as '`.word`'. See section 6.31 [Word], page 38.

6.28 .space *size* , *fill*

Emit *size* bytes, each of value *fill*. Both *size* and *fill* are absolute expressions. If the comma and *fill* are omitted, *fill* is assumed to be zero.

6.29 .stabd, .stabn, .stabs

There are three directives that begin `.stab`.... All emit symbols, for use by symbolic debuggers. The symbols are not entered in `as`' hash table: they cannot be referenced elsewhere in the source file. Up to five fields are required:

string This is the symbol's name. It may contain any character except '`\000`', so is more general than ordinary symbol names. Old debuggers used to code arbitrarily complex structures into symbol names using this technique.

type An absolute expression. The symbol's `n_type` is set to the low 8 bits of this expression. Any bit pattern is permitted, but `ld` and debuggers will choke on silly bit patterns.

other An absolute expression. The symbol's `n_other` is set to the low 8 bits of this expression.

desc An absolute expression. The symbol's `n_desc` is set to the low 16 bits of this expression.

value An absolute expression which becomes the symbol's `n_value`.

If a warning is detected while reading the `.stab`... statement the symbol has probably already been created and you will get a half-formed symbol in your object file. This is compatible with earlier assemblers (!)

.stabd *type* , *other* , *desc*

The "name" of the symbol generated is not even an empty string. It is a null pointer, for compatibility. Older assemblers used a null pointer so they didn't waste space in object files with empty strings.

The symbol's `n_value` is set to the location counter, relocatably. When your program is linked, the value of this symbol will be where the location counter was when the `.stabd` was assembled.

.stabn *type* , *other* , *desc* , *value*

The name of the symbol is set to the empty string "".

.stabs *string* , *type* , *other* , *desc* , *value*

6.30 .text *subsegment number*

Tells `as` to assemble the following statements onto the end of the text subsegment numbered *segment number*, which is an absolute expression.

6.31 .word *expressions*

Expect zero or more *expressions*, of any segment, separated by commas. For each expression, emit a 16-bit number that will, at run time, be the value of that expression. The byte order of the expression depends on what kind of computer will run the program.

6.32 Deprecated Directives

One day these directives won't work. They are included for compatibility with older assemblers.

```
.abort
.file
.line
```

7. Machine Dependent Features

7.1 Vax

7.1.1 Floating Point

Conversion of flonums to floating point is correct, and compatible with previous assemblers. Rounding is towards zero if the remainder is exactly half the least significant bit.

`D`, `F`, `G` and `H` floating point formats are understood.

Immediate floating literals (*e.g.* '`S'$6.9`') are rendered correctly. Again, rounding is towards zero in the boundary case.

The floating point formats generated by directives are these.

`.float`
`.ffloat` `F` format floating point numbers.
`.double`
`.dfloat` `D` format floating point numbers.
`.gfloat` `G` format floating point numbers.
`.hfloat` `H` format floating point numbers.

7.1.2 Machine Directives

The Vax version of the assembler supports four pseudo-ops for generating Vax floating point constants.

7.1.2.1 .dfloat *flonums*

Expect zero or more flonums, separated by commas. Assemble Vax d format floating point constants.

7.1.2.2 .ffloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax f format floating point constants.

7.1.2.3 .gfloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax g format floating point constants.

7.1.2.4 .hfloat *flonums*

Expect zero or more flonums, separated by commas. Assembles Vax h format floating point constants.

7.1.3 Opcodes

All DEC mnemonics are supported. Beware that `case`... instructions have exactly 3 operands. The dispatch table that follows the `case`... instruction should be made with `.word` statements. This is compatible with all un*x assemblers we know of.

7.1.4 Branch Improvement

Certain pseudo opcodes are permitted. They are for branch instructions. They expand to the shortest branch instruction that will reach the target. Generally these mnemonics are made by substituting '`j`' for '`b`' at the start of a DEC mnemonic. This feature is included both for compatibility and to help compilers. If you don't need this feature, don't use these opcodes. Here are the mnemonics, and the code they can expand into.

`jbsb` '`Jsb`' is already an instruction mnemonic, so we chose '`jbsb`'.

(byte displacement)

```
bsbb ...
```

(word displacement)

```
bsbw ...
```

(long displacement)

```
jsb ...
```

`jbr`

`jr` Unconditional branch.

(byte displacement)

```
brb ...
```

(word displacement)

```
brw ...
```

(long displacement)

```
jmp ...
```

`j`*COND* *COND* may be any one of the conditional branches `neq nequ eql eqlu gtr geq lss gtru lequ vc vs gequ cc lssu cs`. *COND* may also be one of the bit test's `bs bc bss bcs bsc bcc bssi bcci lbs lbc`. *NOTCOND* is the opposite condition to *COND*.

(byte displacement)

```
b COND ...
```

(word displacement)

```
b UNCOND foo ; brw ... ; foo:
```

(long displacement)

```
b UNCOND foo ; jmp ... ; foo:
```

`jacb`*X* *X* may be one of `b d f g h l w`.

(word displacement)

```
OPCODE ...
```

(long displacement)

```
OPCODE ..., foo ; brb bar ; foo:  jmp ... ; bar:
```

`jaob`*YYY* *YYY* may be one of `lss leq`.

`jsob`*ZZZ* *ZZZ* may be one of `geq gtr`.

(byte displacement)

```
OPCODE ...
```

(word displacement)

```
OPCODE ..., foo ; brb bar ; foo:  brw destination ; bar:
```

(long displacement)

```
OPCODE ..., foo ; brb bar ; foo:  jmp destination ; bar:
```

`aobleq`
`aoblss`
`sobgeq`
`sobgtr`

(byte displacement)
OPCODE ...
(word displacement)
OPCODE ..., `foo ; brb bar ; foo:` `brw` *destination* `; bar:`
(long displacement)
OPCODE ..., `foo ; brb bar ; foo:` `jmp` *destination* `; bar:`

7.1.5 operands

The immediate character is '`$`' for Un*x compatibility, not '`#`' as DEC writes it.

The indirect character is '`*`' for Un*x compatibility, not '`@`' as DEC writes it.

The displacement sizing character is '`` ` ``' (an accent grave) for Un*x compatibility, not '`^`' as DEC writes it. The letter preceding '`` ` ``' may have either case. '`G`' is not understood, but all other letters (`b i l s w`) are understood.

Register names understood are `r0 r1 r2 ... r15 ap fp sp pc`. Any case of letters will do.

For instance

```
tstb *w`$4(r5)
```

Any expression is permitted in an operand. Operands are comma separated.

There is some bug to do with recognizing expressions in operands, but I forget what it is. It is a syntax clash because () is used as an address mode and to encapsulate sub-expressions.

7.2 68020

7.2.1 Floating Point

The floating point code is not well tested, and may have subtle bugs in it.

X and P format floating literals are not supported. Feel free to add the code yourself.

The floating point formats generated by directives are these.

`.float` `Single` precision floating point constants.
`.double` `Double` precision floating point constants.

7.2.2 Machine Directives

In order to be compatible with the Sun assembler the 68020 assembler understands the following directives.

`.data1` This directive is identical to a `.data 1` directive.
`.data2` This directive is identical to a `.data 2` directive.
`.even` This directive is identical to a `.align 2` directive.
`.skip` This directive is identical to a `.space` directive.

7.2.3 Opcodes

Danger: Several bugs have been found in the opcode table (and fixed). More bugs may exist. The floating point code is especially untested.

The assembler automatically chooses the proper size for branch instructions. Any attempt to force a short displacement will be silently ignored.

The immediate character is '`#`' for Sun compatibility. The line-comment character is '`|`'. If a '`#`' appears at the beginning of a line, it is ignored unless it looks like '`# line file`', in which case it is treated normally.

7.3 32xxx

`as` for the 32xxx computer family has not been written yet.

8. Maintaining the Assembler

[[this chapter is still being built]]

8.1 Design

We had these goals, in descending priority:

Accuracy. For every program composed by a compiler, `as` should emit "correct" code. This leaves some latitude in choosing addressing modes, order of `relocation_info` structures in the object file, *etc.*

Speed, for usual case.
By far the most common use of `as` will be assembling compiler emissions.

Upward compatibility for existing assembler code.
Well ... we don't support bit fields but everything else seems to be upward compatible. Bit fields could be implemented if someone really cared.

Readability.
The code should be maintainable with few surprises.

We assumed that disk I/O was slow and expensive while memory was fast and access to memory was cheap. We expect the in-memory data structures to be less than 10 times the size of the emitted object file. (Contrast this with the C compiler where in-memory structures might be 100 times object file size!) This suggests:

- Try to read the source file from disk only one time. For other reasons, we do keep the entire source file in memory during assembly so this is not a problem. Also the assembly algorithm should only scan the source text once if the compiler composed the text according to a few simple rules.
- Don't emit the object code file, then back up to fill in the front bytes later.
- Build the object file in memory and do direct writes to disk of large buffers.

RMS suggested a one-pass algorithm which seems to work well. By not parsing text during a second pass considerable time is saved on large programs (*e.g.* the sort of C program

It happened that the data structures needed to emit relocation information to the object file were neatly subsumed into the data structures that do backpatching of addresses after pass 1.

Many of the functions began life as re-usable modules, loosely connected. RMS changed this to gain speed. For example, input parsing routines which used to work on pre-sanitized strings now must parse raw data. Hence they have to import knowledge of the assemblers' comment conventions *etc.*

8.2 Deprecated Feature(?)s

We have stopped supporting some features:

- `.org` statements must have **defined** expressions.
- Bit fields (`:` operator) are entirely unsupported.

It might be a good idea to not support these features in a future release:

- `#` should begin a comment, even in column 1.
- Why support the logical line & file concept any more?
- `.stab`... directives should be abandoned in favor of `.gdb`... directives.
- Subsegments are a good candidate for flushing. Depends on which compilers need them I guess.

8.3 Bugs, Ideas, Further Work

Clearly the major improvement is DON'T USE A TEXT-READING ASSEMBLER for the back end of a compiler. It is much faster to interpret binary gobbledygook from a compiler's tables than to ask the compiler to write out human-readable code just so the assembler can parse it back to binary.

Assuming you use `as` for human written programs: here are some ideas:

- Document (here) `APP`.
- Take advantage of knowing no spaces except after opcode to speed up `as`. (Modify `app.c` to flush useless spaces: only keep space/tabs at begin of line or between 2 symbols.)
- Put pointers in this documentation to '`a.out`' documentation.
- Split the assembler into parts so it can gobble direct binary from *e.g.* `cc`. It is silly for`cc` to compose text just so `as` can parse it back to binary.
- Rewrite hash functions: I want a more modular, faster library.

- Clean up LOTS of code.
- Include all the non-'`.c`' files in the maintenance chapter.
- Document flonums.
- Implement flonum short literals.
- Change all talk of expression operands to expression quantities, or perhaps to expression primaries.
- Implement pass 2.
- Whenever a `.text` or `.data` statement is seen, we close of the current frag with an imaginary `.fill 0`. This is because we only have one obstack for frags, and we can't grow new frags for a new subsegment, then go back to the old subsegment and append bytes to the old frag. All this nonsense goes away if we give each subsegment its own obstack. It makes code simpler in about 10 places, but nobody has bothered to do it because C compiler output rarely changes subsegments (compared to ending frags with relaxable addresses, which is common).

8.4 Sources

Here is a list of the source files in the '`as`' directory.

'`app.c`' The pre-processing phase, which deletes comments, handles whitespace, etc.

'`append.c`'
A subroutine to append a string to another string returning a pointer just after the last `char` appended. (JF: All these little routines should probably all be put in one file.)

'`as.c`' Main program of the assembler `as`.

'`atof-m68k.c`'
Turns a flonum into a 68020 literal constant. Machine specific.

'`atof-vax.c`'
Turns a flonum into a Vax constant. Machine specific.

'`error.c`' error() subfunction of *e.g.* the hash routines. (JF: Should be replaced with calls to as_fatal()?)

'`expr.c`' A branch office of '`read.c`'. Understands expressions, primaries. Inside `as`, primaries are called (expression) *operands*. This is confusing, because we also talk (elsewhere) about instruction *operands*. Also, expression operands are called *quantities* explicitly to avoid confusion with instruction operands. What a mess.

'`frags.c`' Implements the **frag** concept. Without frags, finding the right size for branch instructions would be a lot harder.

'`gdb_blocks.c`'
Implement `.gdbbeg`, `.gdbend`, `.gdbblock` statements.

'gdb_file.c'
: Operating system dependent functions to read the file named in a '-G' switch.

'gdb_symbols.c'
: Implement the .gdbsym statement. Remembers all .gdbsym statements then executes them after assembly when gdb symbols are being built.

'gdb.c'
: Some more functions for the GDB dependent stuff. Since GDB really seems to use .stab directives (like various other debuggers), all this stuff might be worth flushing, in my jaundiced view, anyway. (JF)

'hash.c'
: The symbol table, opcode table *etc.* hashing functions.

'hex_value.c'
: Table of values of digits, for use in atoi() type functions. Could probably be flushed by using calls to strtol(), or something similar.

'input-file.c'
: Operating system dependent source file reading routines. Since error messages often say where we are in reading the source file, they live here too. Since Gas is intended to run under GNU and UN*X only, this might be worth flushing. Anyway, almost all C compilers support stdio.

'input-scrub.c'
: Deals with calling the pre-processor and feeding the chunks back the right way. All this input stuff is badly hacked and needs to be re-written. Its been patched too many times already.

'messages.c'
: Operating system independent parts of fatal and warning message reporting.

'output-file.c'
: Operating system dependent functions that write an object file for as. See 'input-file.c' above.

'read.c'
: Implements all the directives of as. Also passing input lines to the machine dependent part of the assembler.

'strstr.c'
: A C library function that isn't in my C library yet.

'subsegs.c'
: Implements subsegments.

'symbols.c'
: Implements symbols.

'vax.c'
: Vax specific parts of as. Also includes the former files 'vax-ins-parse.c', 'vax-reg-parse.c' and 'vip-op.c'.

'write.c'
: Operating system independent functions to emit an object file for as.

'xmalloc.c'
: Implements `malloc()` or bust. Should be combined into some other file somewhere. (misc.c?)

'xrealloc.c'
: Implements `realloc()` or bust. See 'xmalloc.c'.

'atof-generic.c'
: The following files were taken from a machine-independent subroutine library for manipulating floating point numbers and very large integers.

  'atof-generic.c' turns a string into a flonum internal format floating-point number.

'flonum-const.c'
: Some potentially useful floating point numbers in flonum format.

'flonum-copy.c'
: Copies a flonum.

'flonum-multip.c'
: Multiplies two flonums together.

'bignum-copy.c'
: Copies a bignum.

Here is a list of the header files in the source directory. (Warning: This section may not be very accurate. I didn't write the header files; I just report them.) Also note that I think many of these header files could be cleaned up or eliminated.

'a.out.h' Describes the structures used to create the binary header data inside the object file. Perhaps we should use the one in /usr/include ?

'as.h' Defines all the globally useful things, and pulls in <stdio.h> and <assert.h>.

'bignum.h'
: Macros useful for dealing with bignums.

'expr.h' Structure and macros for dealing with expression()

'flonum.h'
: Structure for dealing with floating point numbers. Includes bignum.h

'frags.h' Macro for appending a byte to the current frag.

'hash.h' Structures and function definitions for the hashing functions.

'input-file.h'
: Function headers for the input-file.c functions.

'm-generic.h'
: generic 68020 header file. To be linked to m68k.h on a non-sun3 system.

'`m-sun3.h`'
: 68020 header file for Sun3 workstations. To be linked to m68k.h before compiling on a Sun3 system.

'`m68k-opcode.h`'
: Opcode table for 68020. Should be identical to the one used by gdb, but may contain more mnemonics.

'`m68k.h`'
: A hard- or symbolic- link to either m-generic.h or m-sun3.h depending on which kind of 68020 you are compiling for.

'`md.h`'
: structures and function headers for things defined in the machine dependent part of the assembler.

'`obstack.h`'
: GNU systemwide include file for manipulating obstacks. Since nobody is running under real GNU yet, we include this file.

'`read.h`'
: Macros and function headers for reading in source files.

'`struct-symbol.h`'
: Structure definition and macros for dealing with the gas internal form of a symbol.

'`subsegs.h`'
: structure definition for dealing with the numbered subsegments of the text and data segments.

'`symbols.h`'
: Macros and function headers for dealing with symbols.

'`vax-inst.h`'
: Vax specific file for describing Vax operands and other Vax-ish things.

'`vax-opcode.h`'
: Vax opcode table.

'`write.h`'
: Structure for doing segment fixups.

8.4.1 Test Directory

(Note: The test directory seems to have disappeared somewhere along the line. If you want it, you'll probably have to find a REALLY OLD dump tape. . .)

The '`test/`' directory is used for regression testing. After you modify `as`, you can get a quick go/nogo confidence test by running the new `as` over the source files in this directory. You use a shell script '`test/do`'.

The tests in this suite are evolving. They are not comprehensive. They have, however, caught

hundreds of bugs early in the debugging cycle of **as**. Most test statements in this suite were naturally selected: they were used to demonstrate actual **as** bugs rather than being written *a prioi*.

Another testing suggestion: over 30 bugs have been found simply by running examples from this manual through **as**. Some examples in this manual are selected to distinguish boundary conditions; they are good for testing **as**.

8.4.1.1 Regression Testing

Each regression test involves assembling a file and comparing the actual output of **as** to "known good" output files. Both the object file and the error/warning message file (stderr) are inspected. Optionally **as**' exit status may be checked. Discrepencies are reported. Each discrepency means either that you broke some part of **as** or that the "known good" files are now out of date and should be changed to reflect the new definition of "good".

Each regression test lives in its own directory, in a tree rooted in the directory '`test/`'. Each such directory has a name ending in '`.ret`', where 'ret' stands for REgression Test. The '`.ret`' ending allows `find (1)` to find all regression tests in the tree, without needing to list them explicitly.

Any '`.ret`' directory must contain a file called '`input`' which is the source file to assemble. During testing an object file '`output`' is created, as well as a file '`stdouterr`' which contains the output to both stderr and stderr. If there is a file '`output.good`' in the directory, and if '`output`' contains exactly the same data as '`output.good`', the file '`output`' is deleted. Likewise '`stdouterr`' is removed if it exactly matches a file '`stdouterr.good`'. If file '`status.good`' is present, containing a decimal number before a newline, the exit status of **as** is compared to this number. If the status numbers are not equal, a file '`status`' is written to the directory, containing the actual status as a decimal number followed by newline.

Should any of the '`*.good`' files fail to match their corresponding actual files, this is noted by a 1-line message on the screen during the regression test, and you can use `find (1)` to find any files named '`status`', '`output`' or '`stdouterr`'.

[end of manual]